1. **Explain parallel streams with an example?**

Parallel streams in Java are useful for processing large datasets efficiently by leveraging multiple CPU cores. Here's a real-time example where we process a list of transactions in parallel:

Scenario: Fraud Detection in Banking Transactions
Imagine a banking system that needs to analyse transactions for potential fraud. Using parallel streams, we can speed up the detection process.

```
class Transaction {
    int id;
    double amount;
    boolean isFraudulent;

    Transaction(int id, double amount) {
        this.id = id;
        this.amount = amount;
        this.isFraudulent = amount > 10000; // Example rule: Flag transactions above $10,000
    }

    public boolean isFraudulent() {
        return isFraudulent;
    }

    public int getId() {
        return id;
    }
}

public class ParallelStreamExample {
    public static void main(String[] args) {
        List<Transaction> transactions = IntStream.rangeClosed(1, 1000)
            .mapToObj(i -> new Transaction(i, Math.random() * 20000))
            .collect(Collectors.toList());

        // Parallel processing to detect fraudulent transactions
        List<Transaction> fraudulentTransactions = transactions.parallelStream()
            .filter(Transaction::isFraudulent)
            .collect(Collectors.toList());

        System.out.println("Fraudulent Transactions Count: " + fraudulentTransactions.size());
    }
}
```

```
Why Use Parallel Streams Here?
✅ Improves Performance – Processes transactions concurrently.
✅ Efficient Fraud Detection – Quickly filters large datasets.
✅ Utilizes Multi-Core CPUs – Reduces processing time.
```

2. **How to do the custom memory management techniques in the Java?**

In Java, memory management is largely handled by the **JVM's garbage collector**, but sometimes custom memory management techniques are needed to optimize performance, especially in **high-performance** or **real-time systems**.
Here's a breakdown of **custom memory management techniques** in Java:

🔧 1. **Object Pooling**

What it is:
Reusing objects instead of creating and destroying them frequently, especially when object creation is expensive.

When to use:

- For frequently used objects like database connections, threads, or buffers.
- In game engines, connection managers, or parsers.

Example:

```
class ObjectPool {
    private final List<Reusable> pool = new ArrayList<>();

    public Reusable acquire() {
        if (!pool.isEmpty()) {
            return pool.remove(pool.size() - 1);
        }
        return new Reusable(); // Create new if none are available
    }

    public void release(Reusable obj) {
        pool.add(obj); // Return to pool
    }
}

class Reusable {
    // your reusable object implementation
}
```

🧹 2. **Manual Nulling of References**

What it is:
Explicitly setting references to `null` when they are no longer needed so they become eligible for garbage collection sooner.

Example:

java Copy Edit

```
buffer = null; // helps GC to reclaim memory earlier
```

Note: Not always necessary, but useful in long-lived objects (like in a servlet or cache).

📦 3. Using `SoftReference` / `WeakReference` / `PhantomReference`

These references let you manage memory more delicately:

- `SoftReference<T>`: GC clears only when memory is low — used for memory-sensitive caches.
- `WeakReference<T>`: GC clears eagerly — useful for canonical mappings or listeners.
- `PhantomReference<T>`: Used to determine when an object is about to be collected — mainly for post-cleanup actions.

Example using SoftReference:

```java
SoftReference<byte[]> cache = new SoftReference<>(new byte[1024 * 1024]);

if (cache.get() != null) {
    System.out.println("Using cached object");
} else {
    System.out.println("Recreating object");
    cache = new SoftReference<>(new byte[1024 * 1024]);
}
```

🧊 4. Memory-Mapped Files

Using `MappedByteBuffer` to map files directly into memory, useful for large files and I/O efficiency.

```java
FileChannel fc = new RandomAccessFile("data.txt", "rw").getChannel();
MappedByteBuffer buffer = fc.map(FileChannel.MapMode.READ_WRITE, 0, fc.size());
```

🚀 5. Custom Allocators (Off-Heap Memory)

Using libraries like:

- `sun.misc.Unsafe` (dangerous, non-standard)
- `ByteBuffer.allocateDirect()` – to allocate memory outside JVM heap (off-heap)
- **Third-party libs** like **Netty's PooledByteBufAllocator**

Why off-heap?
Avoids GC overhead for large, persistent data (e.g., caches, game engines, databases).

```java
ByteBuffer buffer = ByteBuffer.allocateDirect(1024); // off-heap
```

🧠 6. Pre-sizing Collections

Avoid resizing overhead by pre-allocating the required size.

```java
List<String> list = new ArrayList<>(1000); // Avoid internal resizing
```

🔄 7. Custom Garbage Collection Tuning

You can configure the JVM's memory behavior using flags:

```bash
-Xms512m -Xmx1024m -XX:+UseG1GC
```

And even **profile** memory with tools like:

- VisualVM
- JConsole
- YourKit
- Eclipse MAT

3. **How to restrict the access of immutable classes from reflection?**

Preventing reflection from modifying immutable classes in Java requires additional security measures beyond standard encapsulation. Here are some strategies:

🔹 **1. Use a Security Manager (Deprecated in Java 17+)**

- The SecurityManager can restrict reflective access to private fields.

- Example:

```
@Override
public void checkPackageAccess(String pkg) {
    if (pkg.equals("java.lang.reflect")) {
        throw new SecurityException("Reflection is not allowed!");
    }
}
```

- Note: SecurityManager is deprecated in Java 17+, so alternative approaches are recommended.

🔹 **2. Use Sealed Classes (Java 17+)**

- Sealed classes restrict inheritance, preventing unauthorized modifications.

- Example:

```
public sealed class ImmutablePerson permits AllowedSubclass {
    private final String name;
    private final int age;

    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

4. **Explain solid principles with examples?**

The SOLID principles are a set of five design principles in object-oriented programming that help developers create more maintainable, understandable, and flexible software. These principles were introduced by Robert C. Martin, also known as Uncle Bob. SOLID stands for:

- **Single Responsibility Principle (SRP)**
- **Open/Closed Principle (OCP)**
- **Liskov Substitution Principle (LSP)**
- **Interface Segregation Principle (ISP)**
- **Dependency Inversion Principle (DIP)**

Single Responsibility Principle (SRP)
The Single Responsibility Principle states that a class should have only one reason to change, meaning it should have only one responsibility.

❌ **Bad:**

java Copy Edit

```java
class Invoice {
    void calculateTotal() { /* logic */ }
    void printInvoice() { /* logic */ }
    void saveToDatabase() { /* logic */ }
}
```

✅ **Good:**

java Copy Edit

```java
class Invoice {
    void calculateTotal() { /* logic */ }
}

class InvoicePrinter {
    void print(Invoice invoice) { /* logic */ }
}

class InvoiceRepository {
    void save(Invoice invoice) { /* logic */ }
}
```

✅ 2. Open/Closed Principle (OCP)

Definition: Software entities should be open for extension but closed for modification.

❌ **Bad:**

java Copy Edit

```java
class DiscountCalculator {
    double calculate(String type) {
        if (type.equals("gold")) return 0.2;
        else if (type.equals("silver")) return 0.1;
        return 0.0;
    }
}
```

✅ **Good (via polymorphism):**

```java
interface Discount {
    double getDiscount();
}

class GoldDiscount implements Discount {
    public double getDiscount() { return 0.2; }
}

class SilverDiscount implements Discount {
    public double getDiscount() { return 0.1; }
}

class DiscountCalculator {
    public double calculate(Discount discount) {
        return discount.getDiscount();
    }
}
```

✅ 3. Liskov Substitution Principle (LSP)

Definition: Subtypes should be substitutable for their base types without altering the correctness of the program.

❌ **Bad:**

```java
class Bird {
    void fly() {}
}

class Ostrich extends Bird {
    void fly() {
        throw new UnsupportedOperationException(); // violates LSP
    }
}
```

✅ Good:

```java
interface Bird {}

interface FlyingBird extends Bird {
    void fly();
}

class Sparrow implements FlyingBird {
    public void fly() { /* fly logic */ }
}

class Ostrich implements Bird {
    // does not fly
}
```

✅ 4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on methods they do not use.

❌ Bad:

```java
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    public void work() {}
    public void eat() {} // Irrelevant for Robot
}
```

✅ Good:

```java
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Robot implements Workable {
    public void work() {}
}
```

🔁 Dependency Inversion Principle (DIP)

Definition:

> High-level modules should not depend on low-level modules. Both should depend on **abstractions**.

🎯 Real-World Scenario

Suppose you're building a **notification system** that sends messages via **Email** or **SMS**.

❌ Without DIP (Tightly Coupled)

java Copy Edit

```java
class EmailService {
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

class Notification {
    private EmailService emailService = new EmailService();

    public void notifyUser(String message) {
        emailService.send(message);
    }
}
```

❗ Problem:

- `Notification` is tightly coupled to `EmailService`.
- Adding `SMSService` requires modifying `Notification` — violates Open/Closed Principle too.

✅ With DIP (Using Interface Abstraction)

Step 1: Create an abstraction

```java
interface MessageService {
    void send(String message);
}
```

Step 2: Implement Email and SMS services

```java
class EmailService implements MessageService {
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

class SMSService implements MessageService {
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
```

🔶 3. High-Level Module Depends on the Interface

```java
public class NotificationSender {
    private final MessageService messageService;

    // Constructor Injection
    public NotificationSender(MessageService messageService) {
        this.messageService = messageService;
    }

    public void send(String message) {
        messageService.sendMessage(message);
    }
}
```

🔶 4. Usage Example (Main Class)

```java
public class Main {
    public static void main(String[] args) {
        // Use Email service
        MessageService emailService = new EmailService();
        NotificationSender emailNotifier = new NotificationSender(emailService);
        emailNotifier.send("Welcome to our newsletter!");

        // Use SMS service
        MessageService smsService = new SMSService();
        NotificationSender smsNotifier = new NotificationSender(smsService);
        smsNotifier.send("Your OTP is 987654");
    }
}
```

🎯 Why This Follows DIP

- **High-level module** `NotificationSender` does **not** depend on `EmailService` or `SMSService` directly.
- It depends on the **abstraction** `MessageService`.
- This makes the system **extensible**, **testable**, and **maintainable**.

5. **Explain the contract between design patterns and solid principles?**

The SOLID principles **provide guidelines for writing** maintainable, scalable, and flexible **object-oriented software, while** design patterns **offer** reusable solutions **to common software design problems. The** contract between them **lies in how** design patterns embody SOLID principles **to create robust software architectures.**

SOLID Principle	Associated Design Patterns	How They Align
Single Responsibility Principle (SRP)	**Factory Method, Repository Pattern**	Ensures classes have a **single responsibility**, reducing complexity
Open-Closed Principle (OCP)	**Strategy, Decorator, Template Method**	Allows **extending functionality** without modifying existing code
Liskov Substitution Principle (LSP)	**Factory Method, Template Method**	Ensures **subclasses can replace parent classes** without breaking functionality
Interface Segregation Principle (ISP)	**Adapter, Facade**	Avoids forcing classes to **implement unused methods**, improving modularity
Dependency Inversion Principle (DIP)	**Dependency Injection, Abstract Factory**	Promotes **loose coupling** by depending on abstractions rather than concrete implementations

6. **How to handle if consumer is unable to match the speed of producer in Kafka?**

Increase Consumer Parallelism: Scale horizontally by adding more consumer instances within the same consumer group. Kafka will automatically balance the partitions across them, helping to speed up consumption.

Optimize Message Processing: Use asynchronous processing with worker threads to parallelize handling. If possible, process messages in batches instead of one by one.

Enable Backpressure Mechanisms: Introduce flow control mechanisms in the producer to slow down production when consumers lag. Utilize rate limiting in producers or introduce throttling mechanisms.

Use Compact Topic Retention & Consumer Lag Monitoring: Implement a retention policy that ensures old messages are cleaned up. Monitor consumer lag using Kafka metrics and alerting systems to detect bottlenecks early.

Improve Consumer Hardware & Configuration: Increase memory and CPU allocation to ensure consumers have enough power. Optimize disk I/O performance, especially if consumers persist data.

7. **In spring boot when post construct load data?**

In **Spring Boot**, the `@PostConstruct` annotation is used to **run a method immediately after the bean is fully initialized**, but **before** it's made available for use.

✅ When does `@PostConstruct` run?

- After **Spring has instantiated the bean** and **injected all dependencies**.
- It runs **before the bean is put into service** (i.e., before requests hit a controller or service).
- It executes **once per bean**, during application startup.